Blockchain Mini-Project

The decentralized dream team

Members:

PES1UG22CS667 Vaibhaw Verma

PES1UG22CS684 Veluru S L Dheeraj chowdary

PES1UG22CS697 Vineet Goel

How the DeFi Lending & Borrowing DApp Works

Core Concept

The application is a decentralized finance (DeFi) lending platform that allows users to:

- Deposit ETH into a shared liquidity pool (lending)
- Borrow ETH from that same pool (borrowing)
- Repay borrowed ETH with interest (repayment)

Who's Involved

In traditional finance, there would be specific lenders and borrowers with direct relationships. In the DeFi application:

- Lenders: Anyone who deposits ETH into the pool becomes a lender. They're not lending to a specific person, but to the pool itself.
- Borrowers: Anyone who takes ETH from the pool becomes a borrower. They're not borrowing from a specific person, but from the pool.
- Smart Contract: Acts as the trustless intermediary that manages all the funds and enforces the rules.

The Liquidity Pool

The "Pool Liquidity" refers to the total amount of ETH available in the smart contract for borrowing. It's the collective sum of all ETH that has been deposited by lenders, minus what's currently borrowed out.

This is why you can't borrow more than the liquidity - the contract simply doesn't have more ETH to give out.

In the code:

```solidity
require(totalPool >= amount, "Not enough liquidity in pool.");
```

The Technology Stack

Smart Contract (Solidity)

The LendingBorrowing.sol contract is the backbone of the application, containing all the business logic:

1. **Lending Function**: Takes user's ETH and adds it to the pool

```solidity
function lend() external payable {
  require(msg.value > 0, "Must send ETH to lend.");
  lenderDeposits[msg.sender] += msg.value;
  totalPool += msg.value;
}
```

2. **Borrowing Function**: Transfers ETH from pool to borrower and creates a loan record

```solidity
function borrow(uint256 amount) external {
  require(amount > 0, "Borrow amount must be > 0.");
  require(totalPool >= amount, "Not enough liquidity in pool.");
  require(!borrowerLoans[msg.sender].isActive, "Already have an active loan.");

  uint256 dueAmount = amount + (amount * interestRate) / 100;
  borrowerLoans[msg.sender] = Loan({
      amount: amount,
      due: dueAmount,
      isActive: true
  });

  totalPool -= amount;
  payable(msg.sender).transfer(amount);
}
```

3. **Repayment Function**: Takes ETH from borrower and updates loan status

```solidity
function repay() external payable {
  Loan storage loan = borrowerLoans[msg.sender];
  require(loan.isActive, "No active loan.");
  require(msg.value >= loan.due, "Insufficient repayment.");

  totalPool += msg.value;
  loan.isActive = false;
}
```

Frontend (HTML/CSS/JavaScript)

The frontend provides a user interface for interacting with the smart contract:

1. **Connection**: Uses Web3.js to connect to the Ethereum blockchain via MetaMask
2. **Interaction**: Provides buttons and forms to call the smart contract functions
3. **Display**: Shows wallet balance, pool liquidity, borrowed amount, and transaction history

The Tools

- **Ganache**: A personal Ethereum blockchain for development. It provides you with:
  - 10 accounts with pre-funded ETH (usually 100 ETH each)
  - A local blockchain that processes transactions instantly
  - No gas costs (in test mode)
- **Truffle**: A development framework for Ethereum that helps with:
  - Smart contract compilation
  - Deployment to different blockchains (including Ganache)
  - Contract testing
  - Managing migrations
- **MetaMask**: A browser extension wallet that:
  - Manages your Ethereum accounts and private keys
  - Connects your browser to the blockchain
  - Signs transactions when you interact with the DApp
  - Can connect to different networks (including your local Ganache)

The Process Flow

Here's how everything works together:

1. **Setup**:
   - Smart contract is deployed to the Ganache blockchain via Trufle
   - The contract address is stored in your app.js file
   - The contract ABI (interface) is stored in abi.json
2. **Connection**:
   - User opens the DApp website
   - app.js loads and detects MetaMask
   - MetaMask connects to the Ganache blockchain
   - The user's account address is retrieved and displayed
3. **Lending**:
   - User enters an amount of ETH to lend
   - Clicks "Lend" button
   - MetaMask prompts to confirm the transaction
   - Smart contract receives the ETH and updates the pool liquidity
   - Transaction history is updated
4. **Borrowing**:
   - User enters an amount of ETH to borrow
   - Clicks "Borrow" button
   - MetaMask prompts to confirm the transaction (no ETH is sent here)
   - Smart contract checks if there's enough liquidity
   - If approved, the contract sends ETH to the user and creates a loan record
   - Transaction history is updated
5. **Repayment**:
   - User enters an amount of ETH to repay

- Clicks "Repay" button
- MetaMask prompts to confirm the transaction
- Smart contract receives the ETH, clears the loan, and updates the pool
- Transaction history is updated

Advantages Over Traditional Lending

This DeFi lending approach has several advantages over traditional lending:

1. **Permissionless**: Anyone can participate without approval from a central authority
2. **Trustless**: Rules are enforced by code, not by institutions
3. **Transparent**: All transactions are visible on the blockchain
4. **Always Available**: The service runs 24/7 without interruption
5. **Automated**: No manual processing or human middlemen
6. **Reduced Counterparty Risk**: Funds are held by a smart contract, not a potentially insolvent institution

Specific Technical Actions

1. **Web3.js**: Creates the connection between your frontend and the blockchain

```javascript
web3 = new Web3(window.ethereum);
```

2. **Contract Initialization**: Loads the contract interface to interact with

```javascript
contract = new web3.eth.Contract(data.abi, contractAddress);
```

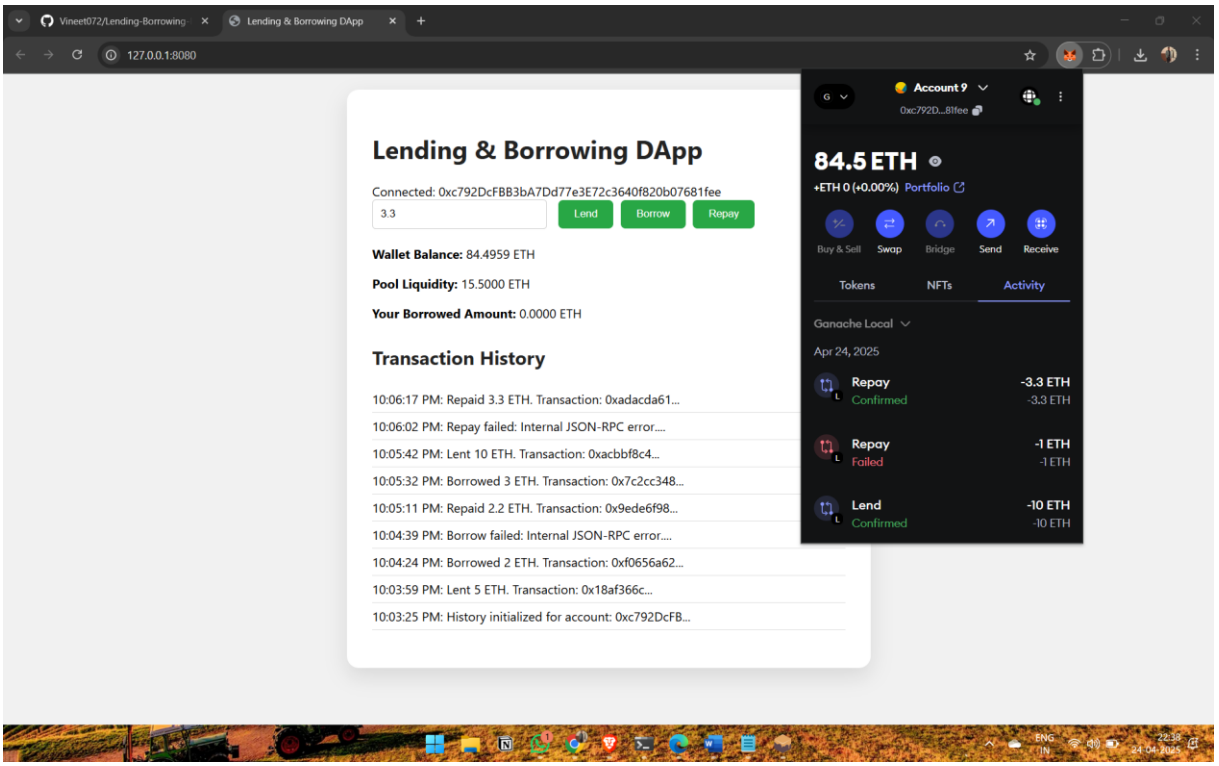
3. **Transaction Sending**: Converts user input to blockchain transactions

```javascript
const tx = await contract.methods.borrow(web3.utils.toWei(value, "ether")).send({
  from: account
});
```

4. **State Reading**: Gets information from the blockchain without transactions

```javascript
const hasLoan = await contract.methods.hasActiveLoan(account).call();
```

The UI looks like:



Github repo link:
https://github.com/Vineet072/Lending-Borrowing-DApp